前言

这个文档的内容**纯手打**，如果想要看更多的干货文章，关注我的公众号：**Java3y**。有更多的原创技术文章和干货！

目前疯狂处于**疯狂**更新PDF中，只要是Java后端的知识，都会有！**欢迎来我公众号催更！**微信搜索：**Java3y**

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！公众号有我的**联系方式**

- 🔥**Java精美脑图**
- 🔥**Java学习路线**
- 🔥**开发常用工具**
- 🔥**精美原创电子书**

在公众号下回复「**888**」即可获取！！

学习不能盲目，跟着我，会让你事半功倍

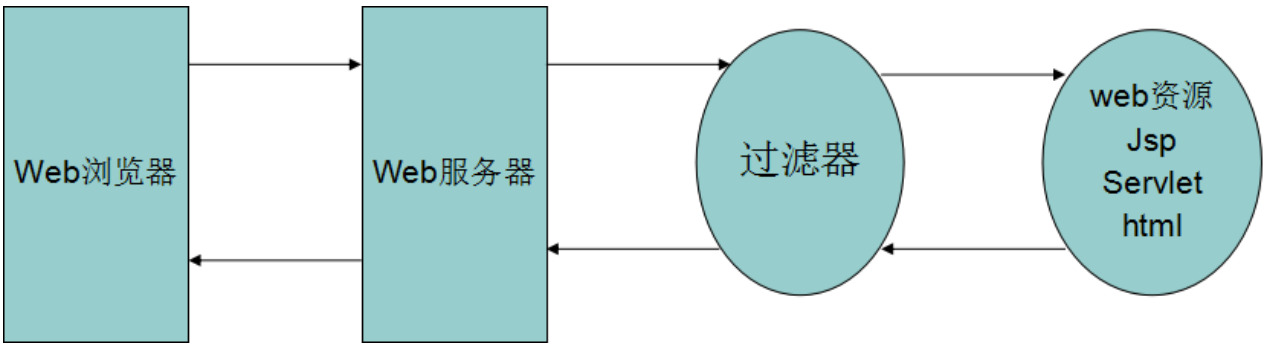
电子书的整理也是挺不容易，如果你觉得有帮助，想要打赏作者，那么可以通过这个收款码打赏我，**金额不重要，心意最重要**。主要是我可以通过这个打赏情况来预计大家对这本电子书的评价，嘻嘻

过滤器入门

1. 什么是过滤器

过滤器是Servlet的高级特性之一，也别把它想得那么高深，只不过是实现Filter接口的Java类罢了！

首先，我们来看看过滤器究竟Web容器的哪处:



从上面的图我们可以发现，**当浏览器发送请求给服务器的时候，先执行过滤器，然后才访问Web的资源。服务器响应Response，从Web资源抵达浏览器之前，也会途径过滤器。。**

我们很容易发现，**过滤器可以比喻成一张滤网**。我们想想现实中的滤网可以做什么：**在泡茶的时候，过滤掉茶叶**。那滤网是怎么过滤茶叶的呢？**规定大小的网孔**，只要网孔比茶叶小，就可以实现过滤了！

引申在Web容器中，过滤器可以做：**过滤一些敏感的字符串【规定不能出现敏感字符串】、避免中文乱码【规定Web资源都使用UTF-8编码】、权限验证【规定只有带Session或Cookie的浏览器，才能访问web资源】等等等，过滤器的作用非常大，只要发挥想象就可以有意想不到的效果**

也就是说：当需要限制用户访问某些资源时、在处理请求时提前处理某些资源、服务器响应的内容对其进行处理再返回、我们就是用过滤器来完成的！

2. 为什么需要用到过滤器

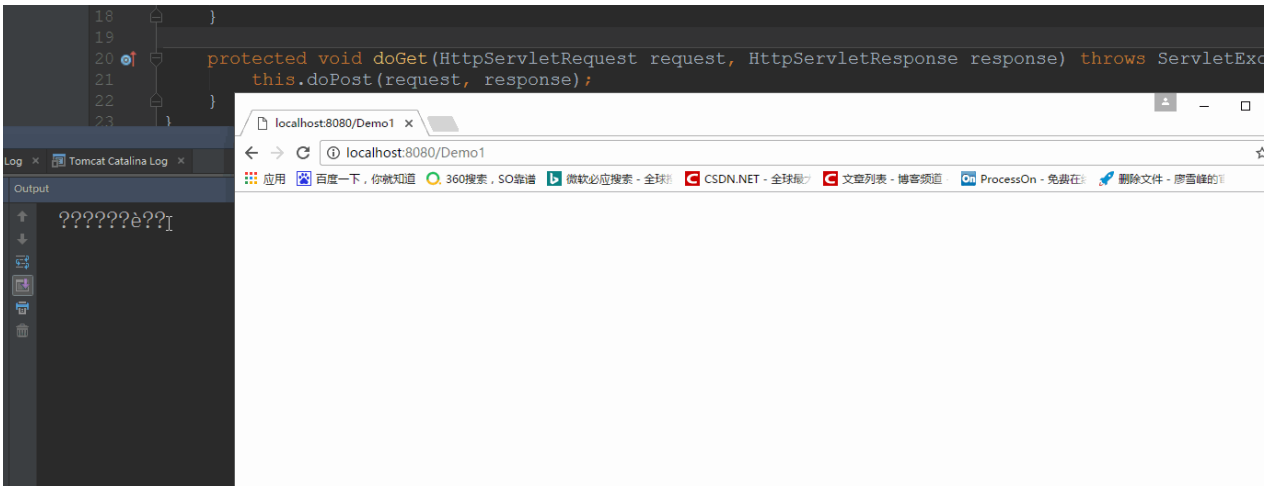
2.1 没有过滤器解决中文乱码问题

如果我没有用到过滤器：**浏览器通过http请求发送数据给Servlet，如果存在中文，就必须指定编码，否则就会乱码！**

jsp页面提交中文数据给Servlet处理

```
<form action="${pageContext.request.contextPath}/Demo1" method="post">

    <input type="text" name="username">

    <input type="submit" value="提交">

</form>
```

Servlet没有指定编码的情况下，获取得到的是乱码



也就是说：如果我每次接受客户端带过来的中文数据，在Serlvet中都要设定编码。这样代码的重复率太高了！！！！

2.2 有过滤器解决中文乱码问题

有过滤器的情况就不一样了：**只要我在过滤器中指定了编码，可以使全站的Web资源都是使用该编码，并且重用性是非常理想的！**

3. 过滤器 API

只要Java类实现了Filter接口就可以称为过滤器！Filter接口的方法也十分简单：

```
public interface Filter {
    void init(FilterConfig var1) throws ServletException;

    void doFilter(ServletRequest var1, ServletResponse var2, FilterChain var3) throws IOException, ServletException;

    void destroy();
}
```

其中init()和destory()方法就不用多说了，他俩跟Servlet是一样的。**只有在Web服务器加载和销毁的时候被执行，只会被执行一次！**

值得注意的是doFilter()方法,**它有三个参数（ServletRequest,ServletResponse,FilterChain），**从前两个参数我们可以发现：**过滤器可以完成任何协议的过滤操作**！

那FilterChain是什么东西呢？我们看看：

```
public interface FilterChain {
    void doFilter(ServletRequest var1, ServletResponse var2) throws IOException, ServletException;
}
```

FilterChain是一个接口，里面又定义了doFilter()方法。这究竟是怎么回事啊？？？？？？

我们可以这样理解：过滤器不单单只有一个，那么我们怎么管理这些过滤器呢？**在Java中就使用了链式结构。把所有的过滤器都放在FilterChain里边，如果符合条件，就执行下一个过滤器（如果没有过滤器了，就执行目标资源）**。

上面的话好像有点拗口，我们可以想象生活的例子：**现在我想在茶杯上能过滤出石头和茶叶出来。石头在一层，茶叶在一层。所以茶杯的过滤装置应该有两层滤网。这个过滤装置就是FilterChain，过滤石头的滤网和过滤茶叶的滤网就是Filter。在石头滤网中，茶叶是属于下一层的，就把茶叶放行，让茶叶的滤网过滤茶叶。过滤完茶叶了，剩下的就是茶（茶就可以比喻成我们的目标资源）**

4. 快速入门

4.1 写一个简单的过滤器

实现Filter接口的Java类就被称作为过滤器

```
  public class FilterDemo1 implements Filter {
      public void destroy() {
      }

      public void doFilter(ServletRequest req, ServletResponse resp,
FilterChain chain) throws ServletException, IOException {

          //执行这一句，说明放行（让下一个过滤器执行，如果没有过滤器了，就执行执行目标资源）
          chain.doFilter(req, resp);
      }

      public void init(FilterConfig config) throws ServletException {

      }
  }
```

4.2 filter部署

过滤器和Servlet是一样的，需要部署到Web服务器上的。

4.2.1 第一种方式：在web.xml文件中配置

`<filter>`用于注册过滤器

```
<filter>
        <filter-name>FilterDemo1</filter-name>
        <filter-class>FilterDemo1</filter-class>
        <init-param>
      <param-name>word_file</param-name>
      <param-value>/WEB-INF/word.txt</param-value>
        </init-param>
</filter>
```

- `<filter-name>`用于**为过滤器指定一个名字**，该元素的内容不能为空。
- `<filter-class>`元素用于指定过滤器的**完整的限定类名。**
- `<init-param>`元素用于为过滤器指定初始化参数，它的子元素指定参数的名字，`<param-value>`指定参数的值。在过滤器中，可以**使用FilterConfig接口对象来访问初始化参数。**

`<filter-mapping>`元素用于**设置一个Filter 所负责拦截的资源。**

一个Filter拦截的资源可**通过两种方式来指定：Servlet 名称和资源访问的请求路径**

```
  <filter-mapping>
      <filter-name>FilterDemo1</filter-name>
      <url-pattern>/*</url-pattern>
  </filter-mapping>
```

- `<filter-name>`子元素用于设置filter的注册名称。**该值必须是在元素中声明过的过滤器的名字**
- **`<url-pattern>`设置 filter 所拦截的请求路径(过滤器关联的URL样式)**
- **`<servlet-name>`指定过滤器所拦截的Servlet名称。**
- `<dispatcher>`指定过滤器所拦截的资源被 Servlet 容器调用的方式，可以是**REQUEST,INCLUDE,FORWARD和ERROR之一，默认REQUEST**。用户可以设置多个`<dispatcher>` 子元素**用来指定 Filter 对资源的多种调用方式进行拦截。**

`<dispatcher>` 子元素可以设置的值及其意义：

- REQUEST：**当用户直接访问页面时，Web容器将会调用过滤器**。如果目标资源是通过RequestDispatcher的include()或forward()方法访问时，那么该过滤器就不会被调用。

- INCLUDE：如果目标资源**是通过RequestDispatcher的include()方法访问时，那么该过滤器将被调用**。除此之外，该过滤器不会被调用。
- FORWARD：如果目标资源是通过**RequestDispatcher的forward()方法访问时，那么该过滤器将被调用**，除此之外，该过滤器不会被调用。
- ERROR：如果目标资源是**通过声明式异常处理机制调用时，那么该过滤器将被调用**。除此之外，过滤器不会被调用。

4.2.2 第二种方式：通过注解配置

```
@WebFilter(filterName = "FilterDemo1",urlPatterns = "/*")
```

上面的配置是"/*"，所有的Web资源都需要途径过滤器

如果想要部分的Web资源进行过滤器过滤则需要指定Web资源的名称即可！

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

5. 过滤器的执行顺序

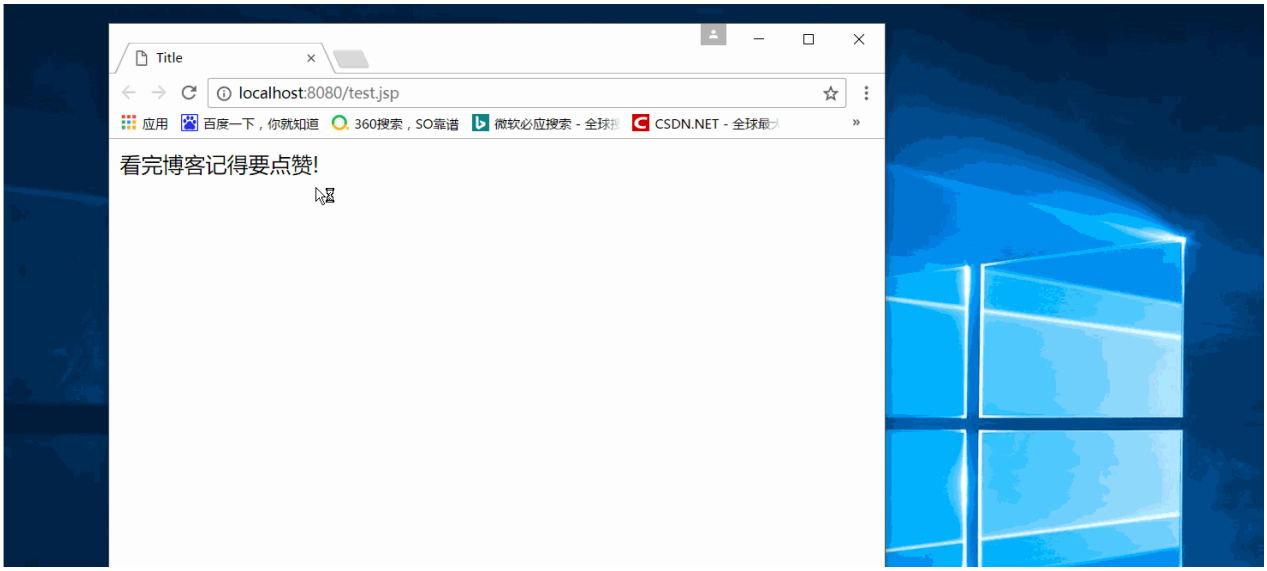
上面已经说过了，过滤器的doFilter()方法是极其重要的，**FilterChain接口是代表着所有的Filter，FilterChain中的doFilter()方法决定着是否放行下一个过滤器执行（如果没有过滤器了，就执行目标资源）**。

5.1 测试一

首先在过滤器的doFilter()中输出一句话，并且调用chain对象的doFilter()方法

```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {

        System.out.println("我是过滤器1");

        //执行这一句，说明放行（让下一个过滤器执行，或者执行目标资源）
        chain.doFilter(req, resp);
    }
```

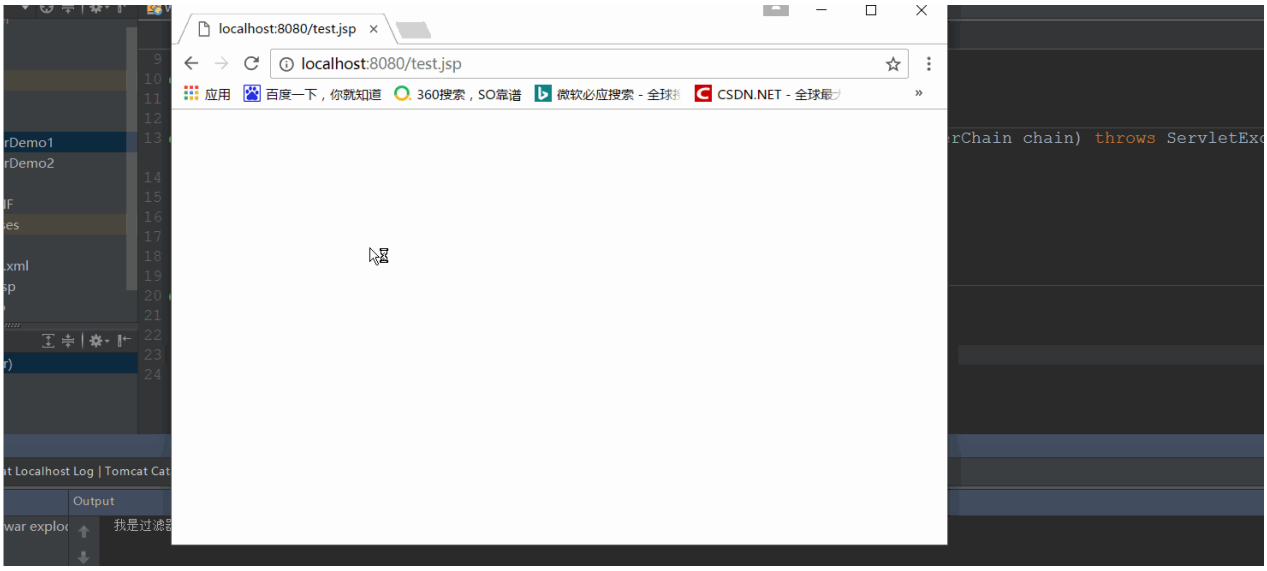
我们来访问一下test.jsp页面：



我们发现test.jsp（我们的目标资源）成功访问到了，并且在服务器上也打印了字符串！

5.2 测试二

我们来试试把`chain.doFilter(req, resp);`这段代码注释了看看！



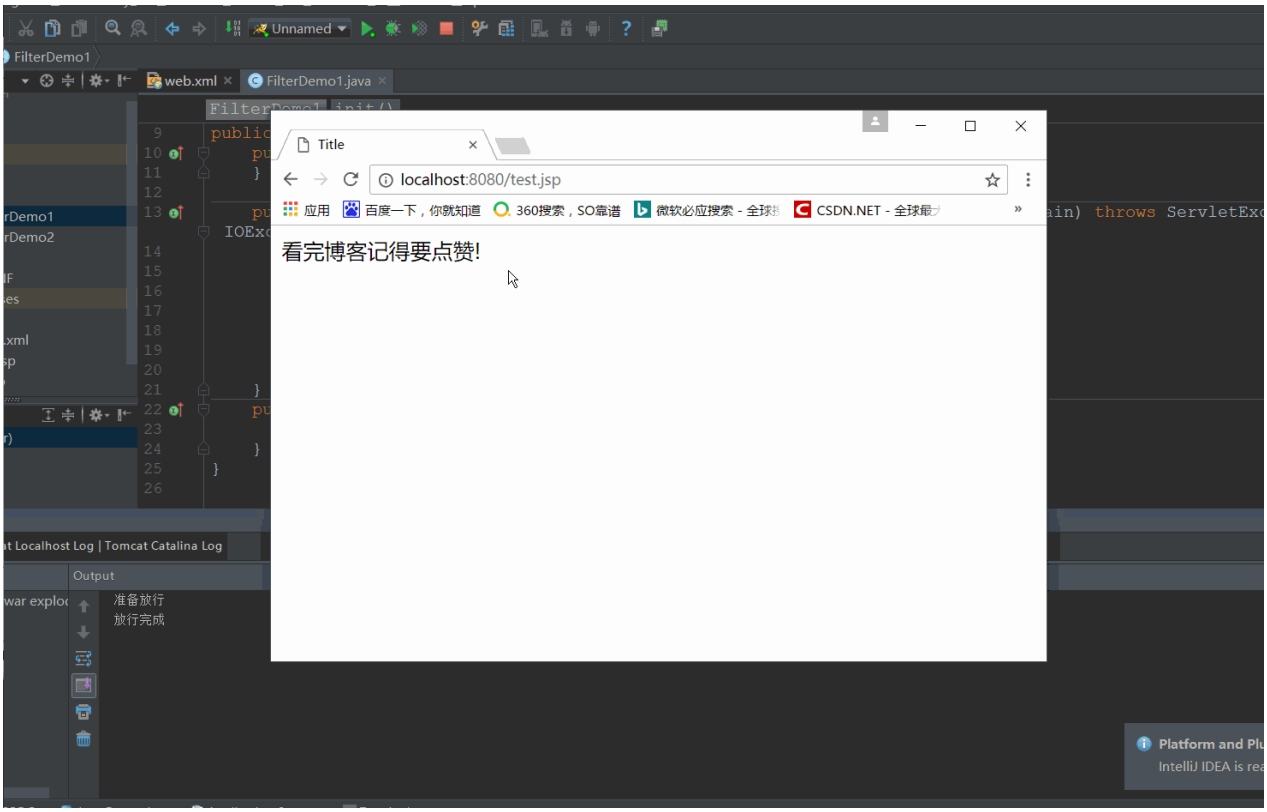
test.jsp页面并没有任何的输出（也就是说，并没有访问到jsp页面）。

5.3 测试三

直接看下面的代码。我们已经知道了”准备放行“会被打印在控制台上和test.jsp页面也能被访问得到，**但“放行完成“会不会打印在控制台上呢?**

```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {

        System.out.println("准备放行");

        //执行这一句，说明放行（让下一个过滤器执行，或者执行目标资源）
        chain.doFilter(req, resp);

        System.out.println("放行完成");
    }
```

答案也非常简单，**肯定会打印在控制台上的**。我们来看看：



注意，它的完整流程顺序是这样的：**客户端发送http请求到Web服务器上，Web服务器执行过滤器，执行到”准备放行“时，就把字符串输出到控制台上，接着执行doFilter()方法，Web服务器发现没有过滤器了，就执行目标资源（也就是test.jsp）。目标资源执行完后，回到过滤器上，继续执行代码，然后输出”放行完成“**

5.4 测试四

我们再多加一个过滤器，看看执行顺序。

过滤器1

```
        System.out.println("过滤器1开始执行");

        //执行这一句，说明放行（让下一个过滤器执行，或者执行目标资源）
        chain.doFilter(req, resp);

        System.out.println("过滤器1开始完毕");
```

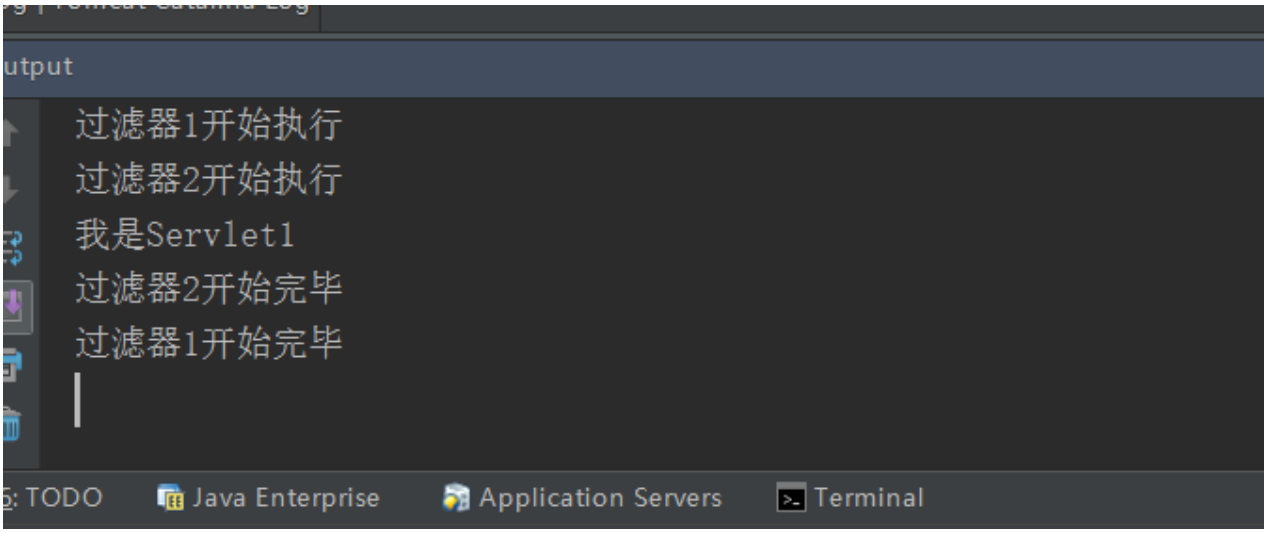
过滤器2

```
        System.out.println("过滤器2开始执行");
        chain.doFilter(req, resp);
        System.out.println("过滤器2开始完毕");
```

Servlet

```
System.out.println("我是Servlet1");
```

当我们访问Servlet1的时候，看看控制台会出现什么：



执行顺序是这样的：**先执行FilterDemo1，放行，执行FilterDemo2，放行，执行Servlet1，Servlet1执行完回到FilterDemo2上，FilterDemo2执行完毕后，回到FilterDemo1上**

注意：过滤器之间的执行顺序看在web.xml文件中mapping的先后顺序的，如果放在前面就先执行，放在后面就后执行！如果是通过注解的方式配置，就比较urlPatterns的字符串优先级

6. Filter简单应用

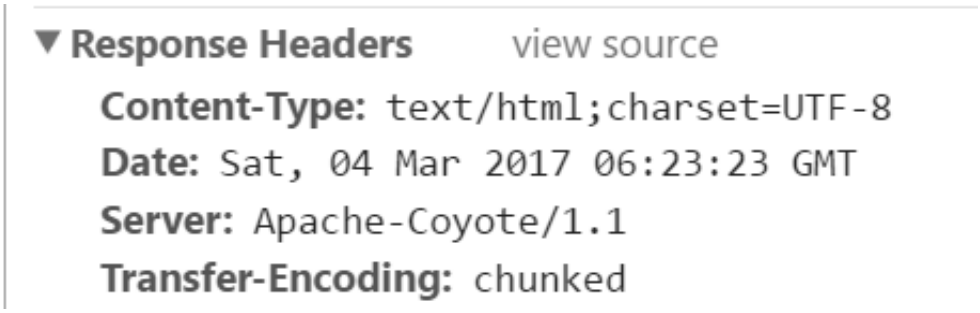
- filter的三种典型应用：
- **1、可以在filter中根据条件决定是否调用chain.doFilter(request, response)方法，即是否让目标资源执行**
- **2、在让目标资源执行之前，可以对request\response作预处理，再让目标资源执行**
- **3、在目标资源执行之后，可以捕获目标资源的执行结果，从而实现一些特殊的功能**

6.1 禁止浏览器缓存所有动态页面

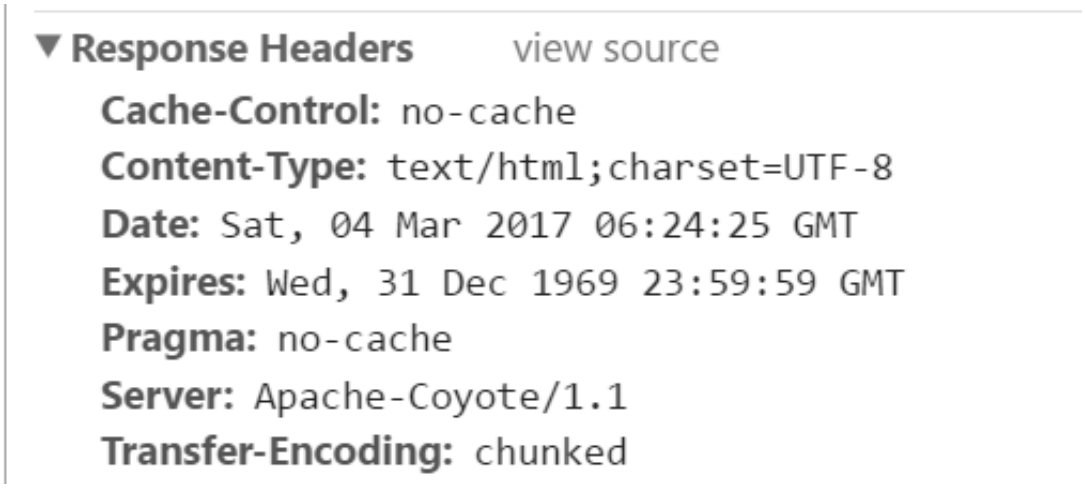
```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {

        //让Web资源不缓存，很简单，设置http中response的请求头即可了！

        //我们使用的是http协议，ServletResponse并没有能够设置请求头的方法，所以要强转成
HttpServletRequest

        //一般我们写Filter都会把他俩强转成Http类型的
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;
```

```
        response.setDateHeader("Expires", -1);
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Pragma", "no-cache");

        //放行目标资源的response已经设置成不缓存的了
        chain.doFilter(request, response);
    }
```

没有过滤之前，响应头是这样的：



过滤之后，响应头是这样的：



6.2 实现自动登陆

实体：

```
    private String username ;
    private String password;


    public User() {
    }

    public User(String username, String password) {
```

```
        this.username = username;
        this.password = password;
    }

  //各种setter和getter
```

集合模拟数据库

```
  public class UserDB {

      private static List<User> users = new ArrayList<>();



      static {
          users.add(new User("aaa", "123"));
          users.add(new User("bbb", "123"));
          users.add(new User("ccc", "123"));
      }

      public static List<User> getUsers() {
          return users;
      }

      public static void setUsers(List<User> users) {
          UserDB.users = users;
      }
  }
```

开发dao

```
public User find(String username, String password) {

  List<User> userList = UserDB.getUsers();

  //遍历List集合，看看有没有对应的username和password
  for (User user : userList) {
    if (user.getUsername().equals(username) &&
user.getPassword().equals(password)) {
      return user;
    }
  }
  return null;
}
```

界面

```
<form action="${pageContext.request.contextPath}/LoginServlet">

    用户名<input type="text" name="username">
    <br>
    密码<input type="password" name="password">
    <br>

    <input type="radio" name="time" value="10">10分钟
    <input type="radio" name="time" value="30">30分钟
    <input type="radio" name="time" value="60">1小时
    <br>

    <input type="submit" value="登陆">

</form>
```

Servlet：

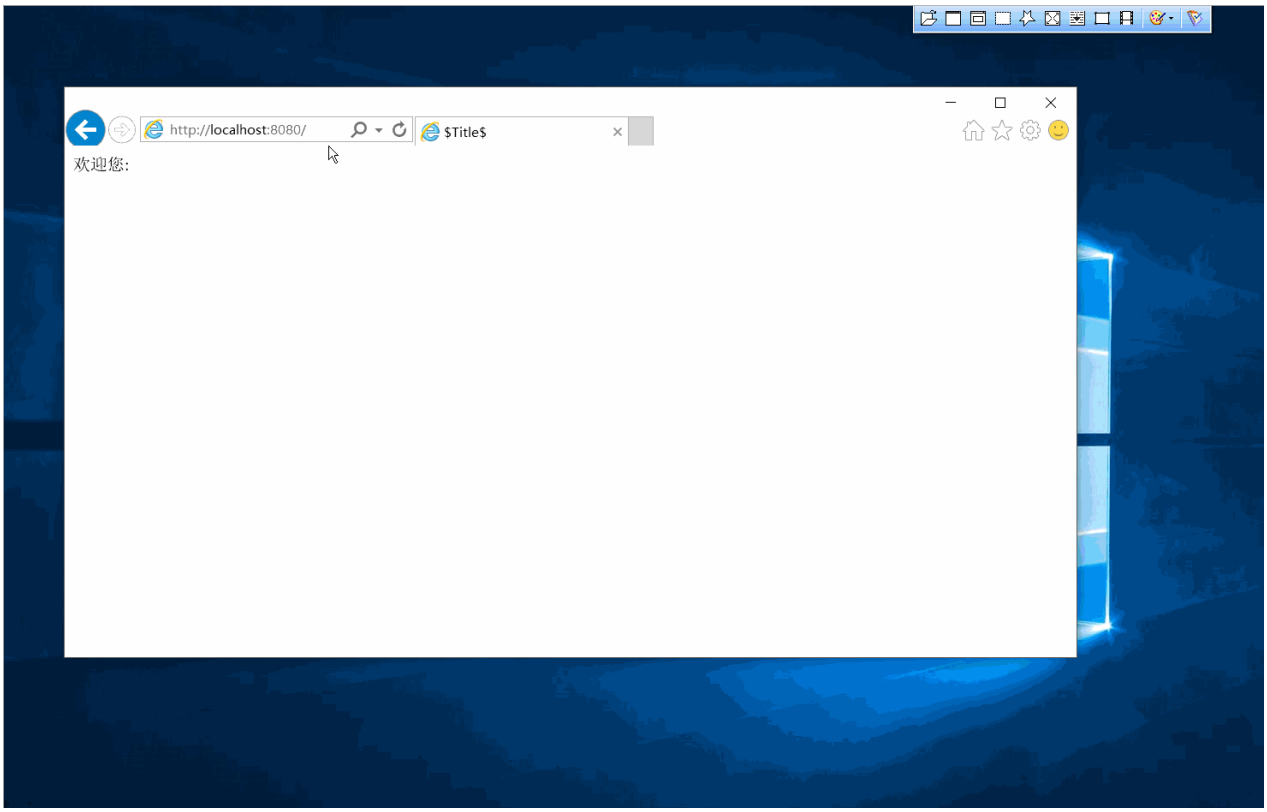
```
        //得到客户端发送过来的数据
        String username = request.getParameter("username");
        String password = request.getParameter("password");

        UserDao userDao = new UserDao();
        User user = userDao.find(username, password);

        if (user == null) {
            request.setAttribute("message", "用户名或密码是错的！");
            request.getRequestDispatcher("/message.jsp").forward(request,
response);
        }

        //如果不是为空，那么在session中保存一个属性
        request.getSession().setAttribute("user", user);
        request.setAttribute("message", "恭喜你，已经登陆了！");

        //如果想要用户关闭了浏览器，还能登陆，就必须要用到Cookie技术了
        Cookie cookie = new Cookie("autoLogin", user.getUsername() + "." +
user.getPassword());

        //设置Cookie的最大声明周期为用户指定的
        cookie.setMaxAge(Integer.parseInt(request.getParameter("time")) * 60);
```

```
        //把Cookie返回给浏览器
        response.addCookie(cookie);

        //跳转到提示页面
        request.getRequestDispatcher("/message.jsp").forward(request,
response);
```

过滤器：

```
        HttpServletResponse response = (HttpServletResponse) resp;
        HttpServletRequest request = (HttpServletRequest) req;

        //如果用户没有关闭浏览器，就不需要Cookie做拼接登陆了
        if (request.getSession().getAttribute("user") != null) {
            chain.doFilter(request, response);
            return;
        }

        //用户关闭了浏览器，session的值就获取不到了。所以要通过Cookie来自动登陆
        Cookie[] cookies = request.getCookies();
        String value = null;
        for (int i = 0; cookies != null && i < cookies.length; i++) {
            if (cookies[i].getName().equals("autoLogin")) {
                value = cookies[i].getValue();
            }
        }

        //得到Cookie的用户名和密码
        if (value != null) {

            String username = value.split("\\.")[0];
            String password = value.split("\\.")[1];

            UserDao userDao = new UserDao();
            User user = userDao.find(username, password);

            if (user != null) {
                request.getSession().setAttribute("user", user);
            }
        }

        chain.doFilter(request, response);
```

效果：



6.2.1改良

我们直接把用户名和密码都放在了Cookie中，这是明文的。懂点编程的人就会知道你的账号了。

于是乎，我们要对密码进行加密！

```
Cookie cookie = new Cookie("autoLogin", user.getUsername() + "." +
md5.md5(user.getPassword()));
```

在过滤器中，加密后的密码就不是数据库中的密码的。所以，我们得**在Dao添加一个功能【根据用户名，找到用户】**

```
    public User find(String username) {
        List<User> userList = UserDB.getUsers();

        //遍历List集合，看看有没有对应的username和password
        for (User user : userList) {
            if (user.getUsername().equals(username)) {
                return user;
            }
        }

        return null;
    }
```

在过滤器中，比较Cookie带过来的md5密码和在数据库中获得的密码（也经过md5）是否相同

```
        //得到Cookie的用户名和密码
        if (value != null) {

            String username = value.split("\\.")[0];
            String password = value.split("\\.")[1];

            //在Cookie拿到的密码是md5加密过的，不能直接与数据库中的密码比较
            UserDao userDao = new UserDao();
            User user = userDao.find(username);

            //通过用户名获得用户信息，得到用户的密码，用户的密码也md5一把

            String dbPassword = md5.md5(user.getPassword());
            //如果两个密码匹配了，就是正确的密码了
            if (password.equals(dbPassword)) {
                request.getSession().setAttribute("user", user);
            }

        }
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

过滤器应用

1. 编码过滤器

目的：**解决全站的乱码问题**

过滤器

```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {

        //将request和response强转成http协议的
        HttpServletRequest httpServletRequest = (HttpServletRequest) req;
        HttpServletResponse httpServletResponse = (HttpServletResponse) resp;

        httpServletRequest.setCharacterEncoding("UTF-8");
        httpServletResponse.setCharacterEncoding("UTF-8");
        httpServletResponse.setContentType("text/html;charset=UTF-8");

        chain.doFilter(httpServletRequest, httpServletResponse);
    }
```

Servlet1中向浏览器回应中文数据，没有出现乱码。

```
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        response.getWriter().write("看完博客点赞！");

    }
```

上面的过滤器是不完善的，因为**浏览器用get方式提交给服务器的中文数据，单单靠上面的过滤器是无法完成的！**

那么我们需要怎么做呢？？我们之前解决get方式的乱码问题是这样的：**使用request获取传递过来的数据，经过ISO 8859-1反编码获取得到不是乱码的数据（传到Servlet上的数据已经被ISO 8859-1编码过了，反编码就可以获取原来的数据），再用UTF-8编码，得到中文数据！**

在Servlet获取浏览器以GET方式提交过来的中文是乱码的根本原因是：getParameter()方法是以ISO 8859-1的编码来获取浏览器传递过来的数据的，得到的是乱码

既然知道了根本原因，那也好办了：**过滤器传递的request对象，使用getParameter()方法的时候，获取得到的是正常的中文数据**

也就是说，sun公司为我们提供的request对象是不够用的，**因为sun公司提供的request对象使用getParameter()获取get方式提交过来的数据是乱码，于是我们要增强request对象（使得getParameter()获取得到的是中文）！**

1.1 增强request对象

增强request对象，我们可以使用包装设计模式！

包装设计模式的五个步骤：

- **1、实现与被增强对象相同的接口**

- **2、定义一个变量记住被增强对象**
- **3、定义一个构造器，接收被增强对象**
- **4、覆盖需要增强的方法**
- **5、对于不想增强的方法，直接调用被增强对象（目标对象）的方法**

sun公司也知道我们可能对request对象的方法不满意，于是提供了HttpServletRequestWrapper类给我们实现（如果实现HttpServletRequest接口的话，要实现太多的方法了！）

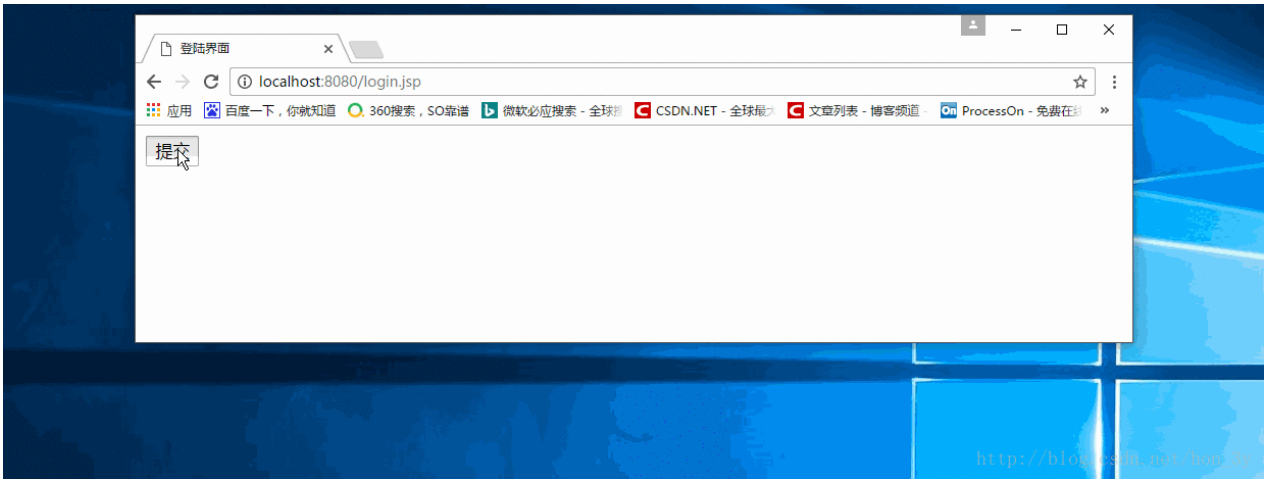
```
class MyRequest extends HttpServletRequestWrapper {

    private HttpServletRequest request;

    public MyRequest(HttpServletRequest request) {
        super(request);
        this.request = request;
    }

    @Override
    public String getParameter(String name) {
        String value = this.request.getParameter(name);

        if (value == null) {
            return null;
        }

        //如果不是get方法的，直接返回就行了
        if (!this.request.getMethod().equalsIgnoreCase("get")) {
            return null;
        }

        try {

            //进来了就说明是get方法，把乱码的数据
            value = new String(value.getBytes("ISO8859-1"),
this.request.getCharacterEncoding());
            return value ;

        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();

            throw new RuntimeException("不支持该编码");
        }

    }
}
```

将被增强的request对象传递给目标资源，那么目标资源使用request调用getParameter()方法的时候，获取得到的就是中文数据，而不是乱码了！

```
        //将request和response强转成http协议的
        HttpServletRequest httpServletRequest = (HttpServletRequest) req;
        HttpServletResponse httpServletResponse = (HttpServletResponse) resp;

        httpServletRequest.setCharacterEncoding("UTF-8");
        httpServletResponse.setCharacterEncoding("UTF-8");
        httpServletResponse.setContentType("text/html;charset=UTF-8");

        MyRequest myRequest = new MyRequest(httpServletRequest);

    //传递给目标资源的request是被增强后的。
        chain.doFilter(myRequest, httpServletResponse);
```

使用get方式传递中文数据给服务器

```
<form action="${pageContext.request.contextPath}/Servlet1" method="get">

    <input type="hidden" name="username" value="中国">


    <input type="submit" value="提交">
</form>
```



2. 敏感词的过滤器

如果用户输入了敏感词（傻b、尼玛、操蛋等等不文明语言时），我们要将这些不文明用于屏蔽掉，替换成符号！

要实现这样的功能也很简单，用户输入的敏感词肯定是在getParameter()获取的，我们在getParameter()得到这些数据的时候，判断有没有敏感词汇，如果有就替换掉就好了！简单来说：也是要增强request对象

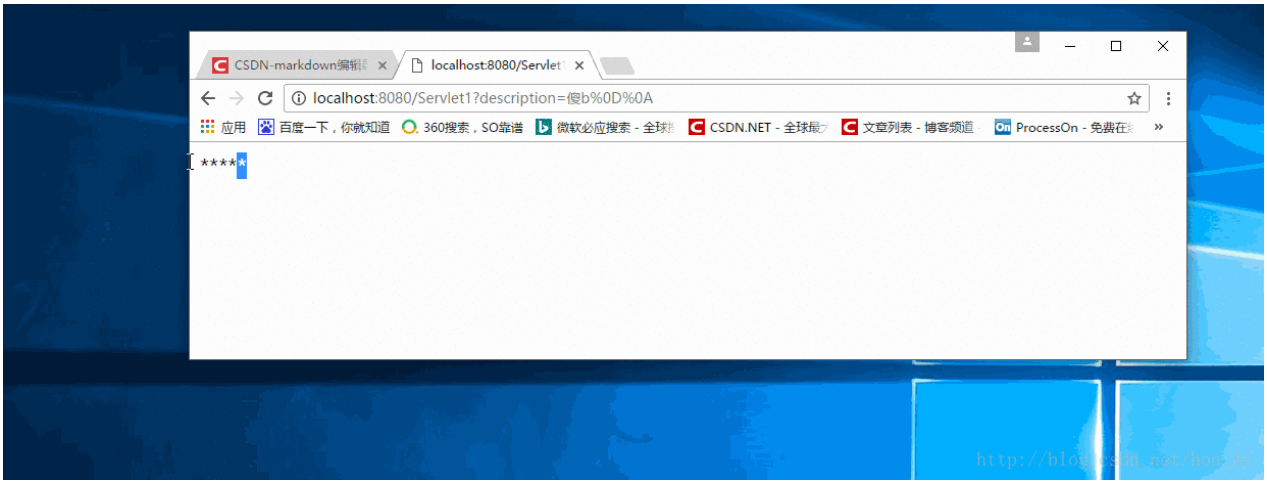
增强request对象

```
class MyDirtyRequest extends HttpServletRequestWrapper {

    HttpServletRequest request;

    //定义一堆敏感词汇
    private List<String> list = Arrays.asList("傻b", "尼玛", "操蛋");

    public MyDirtyRequest(HttpServletRequest request) {
        super(request);
        this.request = request;
    }

    @Override
    public String getParameter(String name) {

        String value = this.request.getParameter(name);

        if (value == null) {
            return null;
        }

        //遍历list集合，看看获取得到的数据有没有敏感词汇
        for (String s : list) {

            if (s.equals(value)) {
                value = "*****";
            }
        }

        return value ;
    }
}
```

开发过滤器:

```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {

        //将request和response强转成http协议的
        HttpServletRequest httpServletRequest = (HttpServletRequest) req;
        HttpServletResponse httpServletResponse = (HttpServletResponse) resp;

        MyDirtyRequest dirtyRequest = new MyDirtyRequest(httpServletRequest);

        //传送给目标资源的是被增强后的request对象
        chain.doFilter(dirtyRequest, httpServletResponse);
    }
```

测试:



3. 压缩资源过滤器

按照过滤器的执行顺序：**执行完目标资源，过滤器后面的代码还会执行。所以，我们在过滤器中可以获取执行完目标资源后的response对象！**

我们知道sun公司提供的response对象调用write()方法，是直接把数据返回给浏览器的。我们要想实现压缩的功能，**write()方法就不能直接把数据写到浏览器上！**

这和上面是类似的，**过滤器传递给目标资源的response对象就需要被我们增强，使得目标资源调用writer()方法的时候不把数据直接写到浏览器上！**

response对象可能会使用PrintWriter或者ServletOutputStream对象来调用writer()方法的，所以我们增强response对象的时候，需要把getOutputSteam和getWriter()重写

```
  class MyResponse extends HttpServletResponseWrapper{
```

```
        HttpServletResponse response;
        public MyResponse(HttpServletResponse response) {
            super(response);
            this.response = response;
        }


        @Override
        public ServletOutputStream getOutputStream() throws IOException {
            return super.getOutputStream();
        }

        @Override
        public PrintWriter getWriter() throws IOException {
            return super.getWriter();
        }
    }
```

接下来，**ServletOutputSteam要调用writer()方法，使得它不会把数据写到浏览器上。这又要我们增强一遍了！**

```
    /*增强ServletOutputSteam，让writer方法不把数据直接返回给浏览器*/
    class MyServletOutputStream extends ServletOutputStream{

        private ByteArrayOutputStream byteArrayOutputStream;

        public MyServletOutputStream(ByteArrayOutputStream
byteArrayOutputStream) {
            this.byteArrayOutputStream = byteArrayOutputStream;
        }

        //当调用write()方法的时候，其实是把数据写byteArrayOutputSteam上
        @Override
        public void write(int b) throws IOException {
            this.byteArrayOutputStream.write(b);

        }
    }
```

PrintWriter对象就好办了，它**本来就是一个包装类，看它的构造方法，我们直接可以把ByteArrayOutputSteam传递给PrintWriter上。**

构造方法摘要

PrintWriter(File file) 使用指定文件创建不具有自动行刷新的新 PrintWriter。
PrintWriter(File file, String csn) 创建具有指定文件和字符集且不带自动刷行新的新 PrintWriter。
PrintWriter(OutputStream out) 根据现有的 OutputStream 创建不带自动行刷新的新 PrintWriter。
PrintWriter(OutputStream out, boolean autoFlush) 通过现有的 OutputStream 创建新的 PrintWriter。
PrintWriter(String fileName) 创建具有指定文件名称且不带自动行刷新的新 PrintWriter。
PrintWriter(String fileName, String csn) 创建具有指定文件名称和字符集且不带自动行刷新的新 PrintWriter。
PrintWriter(Writer out) 创建不带自动行刷新的新 PrintWriter。
PrintWriter(Writer out, boolean autoFlush) 创建新 PrintWriter。


```
    @Override
    public PrintWriter getWriter() throws IOException {
        printWriter = new PrintWriter(new
OutputStreamWriter(byteArrayOutputStream,
this.response.getCharacterEncoding()));

        return printWriter;
    }
```

我们把数据都写在了ByteArrayOutputSteam上了，应该提供方法给外界过去缓存中的数据！

```
    public byte[] getBuffer() {

        try {

            //防止数据在缓存中，要刷新一下！
            if (printWriter != null) {
                printWriter.close();
            }
            if (byteArrayOutputStream != null) {
                byteArrayOutputStream.flush();
                return byteArrayOutputStream.toByteArray();
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
```

```
    }
```

增强response的完整代码

```
class MyResponse extends HttpServletResponseWrapper{

    private ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();

    private PrintWriter printWriter ;

    private HttpServletResponse response;
    public MyResponse(HttpServletResponse response) {
        super(response);
        this.response = response;
    }


    @Override
    public ServletOutputStream getOutputStream() throws IOException {

        //这个的ServletOutputSteam对象调用write()方法的时候，把数据是写在
byteArrayOutputSteam上的
        return new MyServletOutputStream(byteArrayOutputStream);
    }

    @Override
    public PrintWriter getWriter() throws IOException {
        printWriter = new PrintWriter(new
OutputStreamWriter(byteArrayOutputStream,
this.response.getCharacterEncoding()));

        return printWriter;
    }

    public byte[] getBuffer() {

        try {

            //防止数据在缓存中，要刷新一下!
            if (printWriter != null) {
                printWriter.close();
            }
            if (byteArrayOutputStream != null) {
                byteArrayOutputStream.flush();
                return byteArrayOutputStream.toByteArray();
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

过滤器:

```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {


        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;
        MyResponse myResponse = new MyResponse(response);

        //把被增强的response对象传递进去，目标资源调用write()方法的时候就不会直接把数据写
在浏览器上了
        chain.doFilter(request, myResponse);

        //得到目标资源想要返回给浏览器的数据
        byte[] bytes = myResponse.getBuffer();

        //输出原来的大小
        System.out.println("压缩前："+bytes.length);


        //使用GZIP来压缩资源，再返回给浏览器
        ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
        GZIPOutputStream gzipOutputStream = new
GZIPOutputStream(byteArrayOutputStream);
        gzipOutputStream.write(bytes);

        //得到压缩后的数据
        byte[] gzip = byteArrayOutputStream.toByteArray();

        System.out.println("压缩后：" + gzip.length);

        //还要设置头，告诉浏览器，这是压缩数据!
        response.setHeader("content-encoding", "gzip");
        response.setContentLength(gzip.length);
        response.getOutputStream().write(gzip);

    }
```

测试:在Servlet上输出一大段文字

```
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        response.getWriter().write("fdshfidsuhfidusfhuidsfhuidshdsuifhsd" +
                "uifhsduiffffdshfidsuhfidusfhuidsfhuidshdsuif" +
                "hsduifhsduiffffdshfidsuhfidusfhuidsfhuidshd" +
                "suifhsduifhsduiffffdshfidsuhfidusfhuidsfhuidsh" +
                "dsuifhsduifhsduiffffdshfidsuhfidusfhuidsfhuids" +
                "hdsuifhsduifhsduiffffdshfidsuhfidusfhuidsfhuid" +
                "shdsuifhsduifhsduiffdshfidsuhfidusfhuidsfhuids" +
                "hdsuifhsduifhsduiffffdshfidsuhfidusfhuidsfhui" +
                "dshdsuifhsduifhsduiffffdshfidsuhfidusfhuidsfh" +
                "uidshdsuifhsduifhsduiffffdshfidsuhfidusfhuids" +
                "fhuidshdsuifhsduifhsduiffffdshfidsuhfidusfhuid" +
                "sfhuidshdsuifhsduifhsduiffffdshfidsuhfidusfhui" +
                "dsfhuidshdsuifhsduifhsduiffffdshfidsuhfidusfh" +
                "uidsfhuidshdsuifhsduifhsduiffffdshfidsuhfidusf" +
                "huidsfhuidshdsuifhsduifhsduiffffdshfidsuhfidus" +
                "fhuidsfhuidshdsuifhsduifhsduiffffdshfidsuhfid" +
                "usfhuidsfhuidshdsuifhsduifhsduiffffdshfidsuhf" +
                "idusfhuidsfhuidshdsuifhsduifhsd" +
                "uiffffdshfidsuhfidusfhuidsfhuidshdsuifhsduifhsduiffffff");

    }
```

效果:

4. HTML转义过滤器

只要把getParameter()获取得到的数据转义一遍，就可以完成功能了。

增强request

```
class MyHtmlRequest extends HttpServletRequestWrapper{

    private HttpServletRequest request;

    public MyHtmlRequest(HttpServletRequest request) {
        super(request);
        this.request = request;
    }


    @Override
    public String getParameter(String name) {

        String value = this.request.getParameter(name);
        return this.Filter(value);

    }

    public String Filter(String message) {
        if (message == null)
            return (null);

        char content[] = new char[message.length()];
        message.getChars(0, message.length(), content, 0);
        StringBuffer result = new StringBuffer(content.length + 50);
        for (int i = 0; i < content.length; i++) {
            switch (content[i]) {
                case '<':
                    result.append("&lt;");
                    break;
                case '>':
                    result.append("&gt;");
                    break;
                case '&':
                    result.append("&amp;");
                    break;
                case '"':
                    result.append("&quot;");
                    break;
                default:
                    result.append(content[i]);
            }
        }
        return (result.toString());

    }

}
```

过滤器

```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {



        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;
        MyHtmlRequest myHtmlRequest = new MyHtmlRequest(request);

        //传入的是被增强的request!
        chain.doFilter(myHtmlRequest, response);

    }
```

jsp代码：

```
  <form action="${pageContext.request.contextPath}/Servlet1" method="post">


      <input type="hidden" name="username" value="<h1>你好i好<h1>">

      <input type="submit" value="提交">
  </form>
```

Servlet代码：

```
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        String value = request.getParameter("username");
        response.getWriter().write(value);

    }
```

5. 缓存数据到内存中

在前面我们已经做过了，让浏览器不缓存数据【验证码的图片是不应该缓存的】。

现在我们要做的是：**缓存数据到内存中【如果某个资源重复使用，不轻易变化，应该缓存到内存中】**

这个和压缩数据的Filter非常类似的，因为**让数据不直接输出给浏览器，把数据用一个容器（ByteArrayOutputSteam）存起来。如果已经有缓存了，就取缓存的。没有缓存就执行目标资源！**

增强response对象

```
class MyResponse extends HttpServletResponseWrapper {

    private ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();

    private PrintWriter printWriter ;

    private HttpServletResponse response;
    public MyResponse(HttpServletResponse response) {
        super(response);
        this.response = response;
    }


    @Override
    public ServletOutputStream getOutputStream() throws IOException {

        //这个的ServletOutputSteam对象调用write()方法的时候，把数据是写在
byteArrayOutputSteam上的
        return new MyServletOutputStream(byteArrayOutputStream);
    }

    @Override
    public PrintWriter getWriter() throws IOException {
        printWriter = new PrintWriter(new
OutputStreamWriter(byteArrayOutputStream,
this.response.getCharacterEncoding()));

        return printWriter;
```

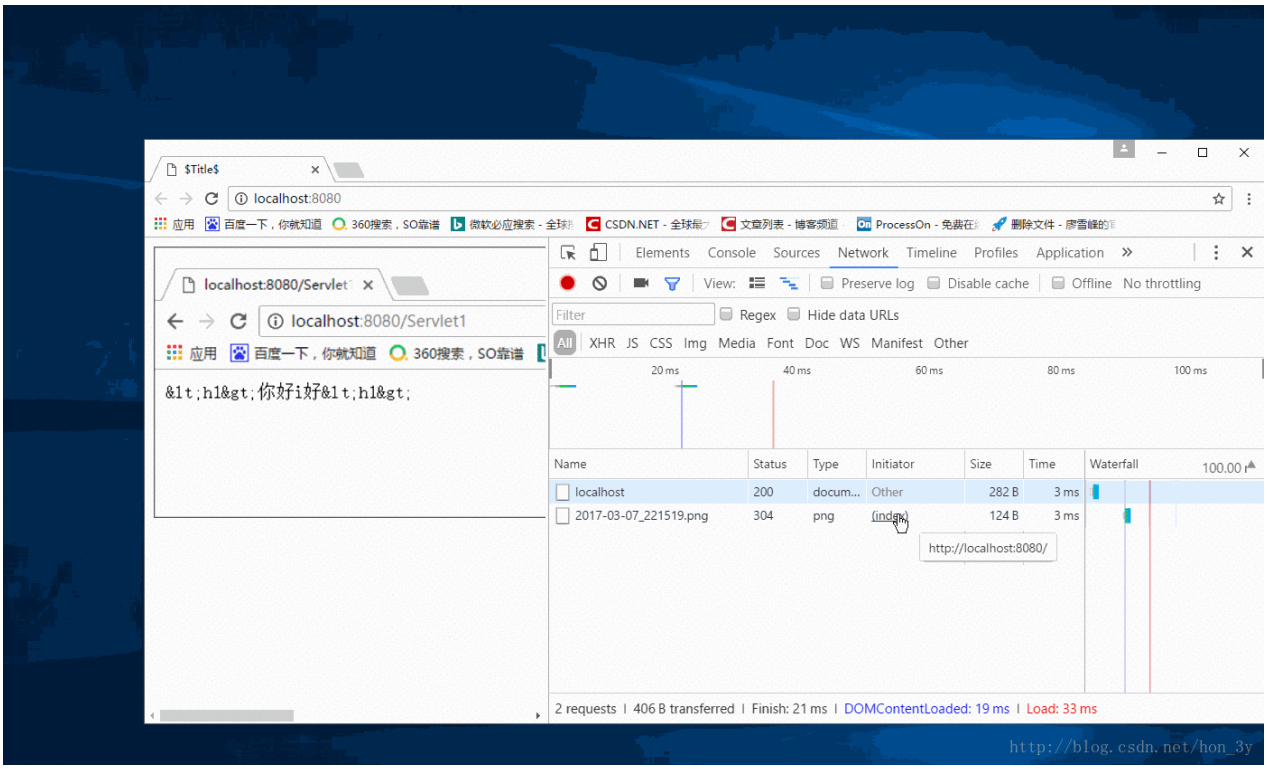
```
    }

    public byte[] getBuffer() {

        try {

            //防止数据在缓存中，要刷新一下!
            if (printWriter != null) {
                printWriter.close();
            }
            if (byteArrayOutputStream != null) {
                byteArrayOutputStream.flush();
                return byteArrayOutputStream.toByteArray();
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
}


//增强ServletOutputSteam，让writer方法不把数据直接返回给浏览器

class MyServletOutputStream extends ServletOutputStream {

    private ByteArrayOutputStream byteArrayOutputStream;

    public MyServletOutputStream(ByteArrayOutputStream byteArrayOutputStream)
{
        this.byteArrayOutputStream = byteArrayOutputStream;
    }

    //当调用write()方法的时候，其实是把数据写byteArrayOutputSteam上
    @Override
    public void write(int b) throws IOException {
        this.byteArrayOutputStream.write(b);

    }
}
```

过滤器

```
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws ServletException, IOException {
```

```
        //定义一个Map集合，key为页面的地址，value为内存的缓存
        Map<String, byte[]> map = new HashMap<>();

        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;

        //得到客户端想要请求的资源
        String uri = request.getRequestURI();
        byte[] bytes = map.get(uri);

        //如果有缓存，直接返回给浏览器就行了，就不用执行目标资源了
        if (bytes != null) {
            response.getOutputStream().write(bytes);
            return ;
        }

        //如果没有缓存，就让目标执行
        MyResponse myResponse = new MyResponse(response);
        chain.doFilter(request, myResponse);

        //得到目标资源想要发送给浏览器的数据
        byte[] b = myResponse.getBuffer();

        //把数据存到集合中
        map.put(uri, b);

        //把数据返回给浏览器
        response.getOutputStream().write(b);


    }
```

尽管是刷新，获取得到的也是从缓存拿到的数据！



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

监听器入门

1. 什么是监听器

> 监听器就是一个**实现特定接口的普通java程序**，这个程序专门用于**监听另一个java对象的方法调用或属性改变**，当被监听对象发生上述事件后，监听器某个方法将立即被执行。

2. 为什么我们要使用监听器?

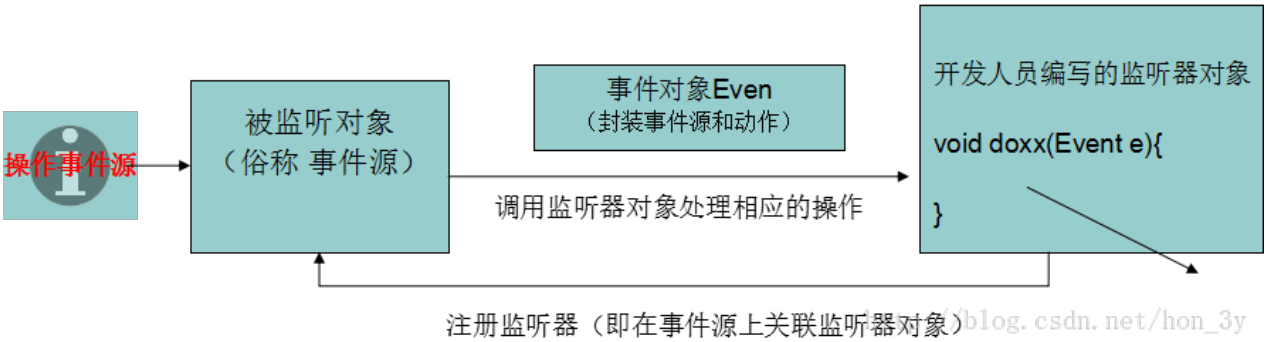
监听器可以**用来检测网站的在线人数，统计网站的访问量等等**!

3. 监听器组件

监听器涉及三个组件：**事件源，事件对象，事件监听器**

当事件源发生某个动作的时候，它会调用事件监听器的方法，并在调用事件监听器方法的时候把事件对象传递进去。

我们在监听器中就可以通过事件对象获取得到事件源，从而对事件源进行操作!



4. 模拟监听器

既然上面已经说了监听器的概念了，监听器涉及三个组件：事件源，事件对象，事件监听器。

我们就写一个对象，被监听器监听

4.1 监听器

监听器定义为接口，**监听的方法需要事件对象传递进来，从而在监听器上通过事件对象获取得到事件源，对事件源进行修改**!

```
/**
 * 事件监听器
 *
 * 监听Person事件源的eat和sleep方法
 */
interface PersonListener{

    void doEat(Event event);
    void doSleep(Event event);
}
```

4.2事件源

事件源是一个Person类，它有eat和sleep()方法。

事件源需要注册监听器(即在事件源上关联监听器对象)

如果**触发了eat或sleep()方法的时候，会调用监听器的方法，并将事件对象传递进去**

```
/**
 *
 * 事件源Person
 *
 * 事件源要提供方法注册监听器(即在事件源上关联监听器对象)
 */

class Person {

    //在成员变量定义一个监听器对象
    private PersonListener personListener ;

    //在事件源中定义两个方法
    public void Eat() {

        //当事件源调用了Eat方法时，应该触发监听器的方法，调用监听器的方法并把事件对象传递进去
        personListener.doEat(new Event(this));
    }

    public void sleep() {

        //当事件源调用了Eat方法时，应该触发监听器的方法，调用监听器的方法并把事件对象传递进去
        personListener.doSleep(new Event(this));
    }

    //注册监听器，该类没有监听器对象啊，那么就传递进来吧。
```

```
    public void registerLister(PersonListener personListener) {
        this.personListener = personListener;
    }

}
```

4.3事件对象

事件对象封装了事件源。监听器可以从事件对象上获取得到事件源的对象(信息)

```
/**
 * 事件对象Even
 *
 * 事件对象封装了事件源
 *
 * 在监听器上能够通过事件对象获取得到事件源
 *
 *
 */
class Event{
    private Person person;

    public Event() {
    }

    public Event(Person person) {
        this.person = person;
    }

    public Person getResource() {
        return person;
    }

}
```

测试

```
    public static void main(String[] args) {

        Person person = new Person();

        //注册监听器()
        person.registerLister(new PersonListener() {
```

```
            @Override
            public void doEat(Event event) {
                Person person1 = event.getResource();
                System.out.println(person1 + "正在吃饭呢！");
            }

            @Override
            public void doSleep(Event event) {
                Person person1 = event.getResource();
                System.out.println(person1 + "正在睡觉呢！");
            }
        });


        //当调用eat方法时，触发事件，将事件对象传递给监听器，最后监听器获得事件源，对事件源进行操作
        person.Eat();
    }
```

```
"X:\Program Files (x86)\Java\jdk1.7.0\bin\java" ...
Person@42552c正在吃饭呢！

Process finished with exit code 0
```


- 事件源：拥有事件
- 监听器：监听事件源所拥有的事件（带事件对象参数的）
- 事件对象：事件对象封装了事件源对象
    - **事件源要与监听器有关系，就得注册监听器【提供方法得到监听器对象】**
    - **触发事件源的事件，实际会提交给监听器对象处理，并且把事件对象传递过去给监听器。**

5.Servle监听器

在Servlet规范中定义了多种类型的监听器，它们**用于监听的事件源分别 ServletContext, HttpSession和ServletRequest这三个域对象**

和其它事件监听器略有不同的是，**servlet监听器的注册不是直接注册在事件源上，而是由WEB容器负责注册**，开发人员**只需在web.xml文件中使用`<listener>`标签配置好监听器**，

5.1监听对象的创建和销毁

HttpSessionListener、ServletContextListener、ServletRequestListener分别监控着Session、Context、Request对象的创建和销毁

- **HttpSessionListener(可以用来收集在线者信息)**
- **ServletContextListener(可以获取web.xml里面的参数配置)**
- **ServletRequestListener**

测试：

```
public class Listener1 implements ServletContextListener,
        HttpSessionListener, ServletRequestListener {

    // Public constructor is required by servlet spec
    public Listener1() {
    }

    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("容器创建了");
    }

    public void contextDestroyed(ServletContextEvent sce) {

        System.out.println("容器销毁了");
    }


    public void sessionCreated(HttpSessionEvent se) {

        System.out.println("Session创建了");
    }

    public void sessionDestroyed(HttpSessionEvent se) {
        System.out.println("Session销毁了");
    }


    @Override
    public void requestDestroyed(ServletRequestEvent servletRequestEvent) {

    }

    @Override
    public void requestInitialized(ServletRequestEvent servletRequestEvent) {

    }
}
```

监听器监听到ServletContext的初始化了，Session的创建和ServletContext的销毁。(服务器停掉，不代表Session就被销毁了。Session的创建是在内存中的，所以没看到Session被销毁了)

5.2监听对象属性变化

ServletContextAttributeListener、HttpSessionAttributeListener、ServletRequestAttributeListener分别监听着Context、Session、Request对象属性的变化

这三个接口中都定义了**三个方法来处理被监听对象中的属性的增加，删除和替换的事件**，同一个事件在这三个接口中对应的方法**名称完全相同，只是接受的参数类型不同。**

- **attributeAdded()**
- **attributeRemoved()**
- **attributeReplaced()**

这里我**只演示Context对象，其他对象都是以此类推的，就不一一测试了。**

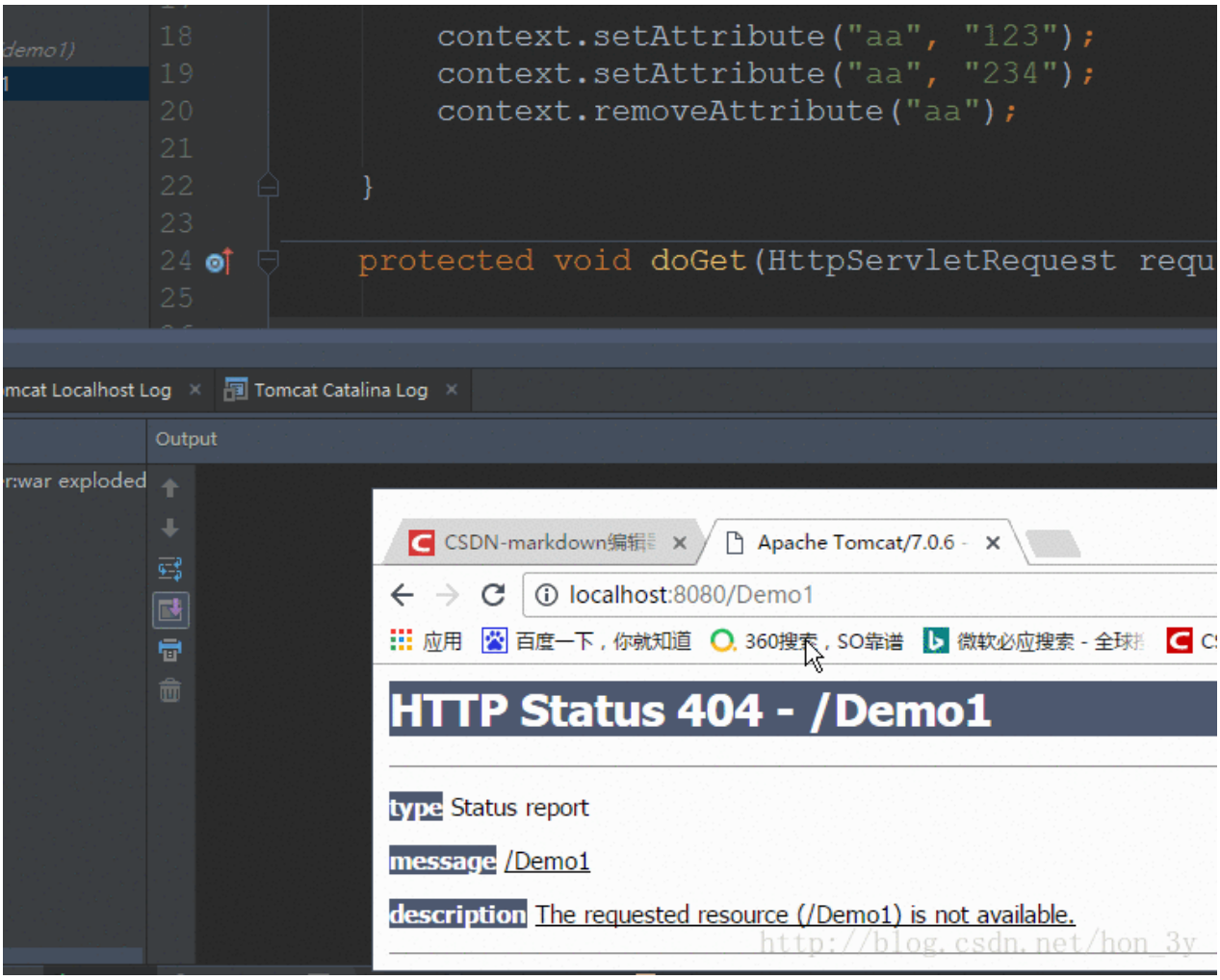
实现ServletContextAttributeListener接口。

```
public class Listener1 implements ServletContextAttributeListener {
```

```
    @Override
    public void attributeAdded(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("Context对象增加了属性");
    }

    @Override
    public void attributeRemoved(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("Context对象删除了属性");

    }

    @Override
    public void attributeReplaced(ServletContextAttributeEvent
servletContextAttributeEvent) {
        System.out.println("Context对象替换了属性");

    }
}
```

测试的Servlet

```
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        ServletContext context = this.getServletContext();

        context.setAttribute("aa", "123");
        context.setAttribute("aa", "234");
        context.removeAttribute("aa");

    }
```



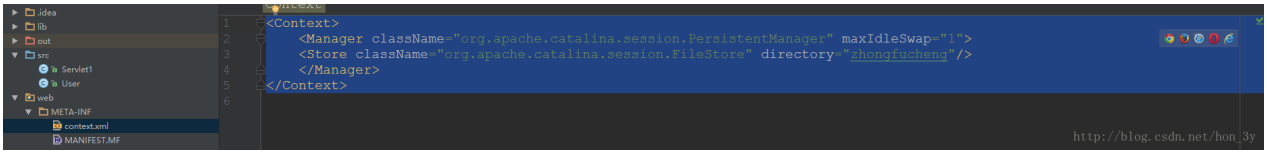
5.3监听Session内的对象

除了上面的6种Listener，还有两种Linstener监听Session内的对象，分别是HttpSessionBindingListener和HttpSessionActivationListener，实现这两个接口并不需要在web.xml文件中注册

- 实现HttpSessionBindingListener接口，**JavaBean 对象可以感知自己被绑定到 Session 中和从 Session 中删除的事件【和HttpSessionAttributeListener的作用是差不多的】**
- 实现HttpSessionActivationListener接口，**JavaBean 对象可以感知自己被活化和钝化的事件（当服务器关闭时，会将Session的内容保存在硬盘上【钝化】，当服务器开启时，会将Session的内容在硬盘式重新加载【活化】）** 。。

想要测试出Session的硬化和钝化，需要修改Tomcat的配置的。在META-INF下的context.xml文件中添加下面的代码：

```
<Context>
  <Manager className="org.apache.catalina.session.PersistentManager"
maxIdleSwap="1">
  <Store className="org.apache.catalina.session.FileStore"
directory="zhongfucheng"/>
  </Manager>
</Context>
```



监听器和事件源

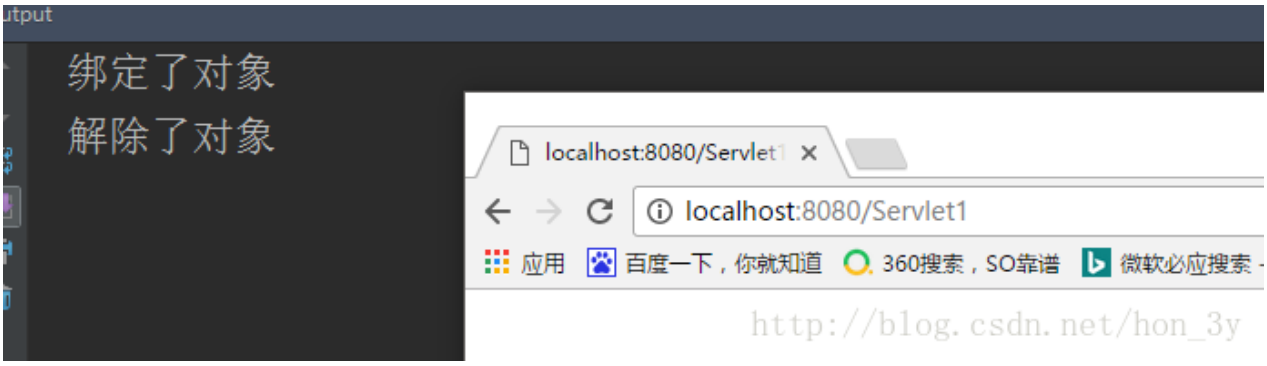
```
/*
* 由于涉及到了将内存的Session钝化到硬盘和用硬盘活化到内存中，所以需要实现Serializable接口
*
* 该监听器是不需要在web.xml文件中配置的。但监听器要在事件源上实现接口
* 也就是说，直接用一个类实现HttpSessionBindingListener和
HttpSessionActivationListener接口是监听不到Session内对象的变化的。
* 因为它们是感知自己在Session中的变化!
* */
public class User implements
HttpSessionBindingListener,HttpSessionActivationListener,Serializable {

    private String username ;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }


    @Override
    public void sessionWillPassivate(HttpSessionEvent httpSessionEvent) {

        HttpSession httpSession = httpSessionEvent.getSession();

        System.out.println("钝化了");

    }

    @Override
```

```
    public void sessionDidActivate(HttpSessionEvent httpSessionEvent) {
        HttpSession httpSession = httpSessionEvent.getSession();
        System.out.println("活化了");

    }
    @Override
    public void valueBound(HttpSessionBindingEvent httpSessionBindingEvent) {

        System.out.println("绑定了对象");
    }
    @Override
    public void valueUnbound(HttpSessionBindingEvent httpSessionBindingEvent)
{
        System.out.println("解除了对象");

    }
}
```

测试代码

```
        User user = new User();
        request.getSession().setAttribute("aaa", user);
        request.getSession().removeAttribute("aaa");
```

效果:



如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

监听器应用

1. 统计网站在线人数

ps:这个可以使用WebSocket来做，但这里讲解的是监听器，所以这里以监听器来举例子。

我们在网站中一般使用Session来标识某用户是否登陆了，如果登陆了，就在Session域中保存相对应的属性。如果没有登陆，那么Session的属性就应该为空。

现在，我们想要统计的是网站的在线人数。我们应该这样做：**我们监听是否有新的Session创建了，如果新创建了Sesssion，那么在线人数就应该+1。这个在线人数是整个站点的，所以应该有Context对象保存。**

大致思路：

- 监听Session是否被创建了
- 如果Session被创建了，那么在Context的域对象的值就应该+1
- 如果Session从内存中移除了，那么在Context的域对象的值就应该-1.

监听器代码：

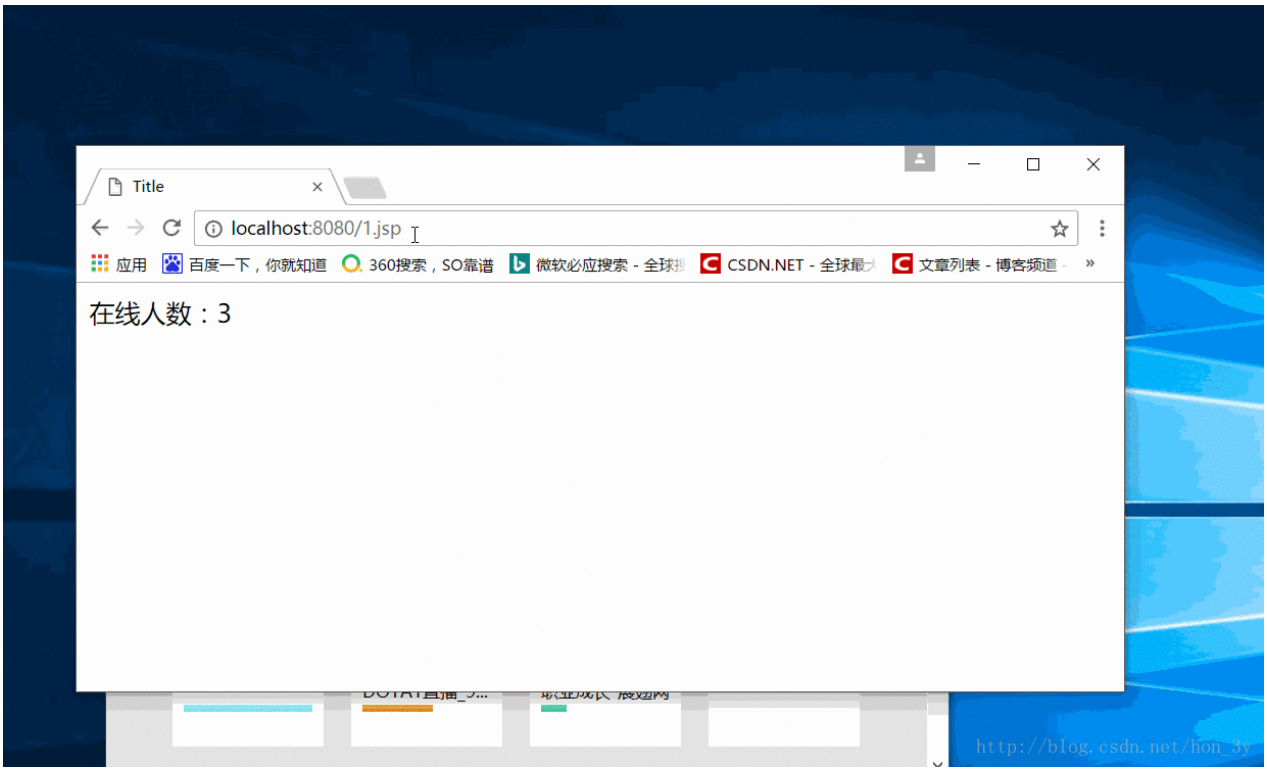
```
  public class CountOnline implements HttpSessionListener {

      public void sessionCreated(HttpSessionEvent se) {

          //获取得到Context对象，使用Context域对象保存用户在线的个数
          ServletContext context = se.getSession().getServletContext();

          //直接判断Context对象是否存在这个域，如果存在就人数+1,如果不存在，那么就将属性设置到Context域中
          Integer num = (Integer) context.getAttribute("num");

          if (num == null) {
              context.setAttribute("num", 1);
          } else {
              num++;
              context.setAttribute("num", num);
          }
      }
      public void sessionDestroyed(HttpSessionEvent se) {

          ServletContext context = se.getSession().getServletContext();
          Integer num = (Integer) se.getSession().getAttribute("num");

          if (num == null) {
              context.setAttribute("num", 1);
          } else {
              num--;
              context.setAttribute("num", num);
          }
      }
  }
```

显示页面代码：

```
在线人数：${num}
```

我们每使用一个浏览器访问服务器，都会新创建一个Session。那么网站的在线人数就会+1。

使用同一个页面刷新，还是使用的是那个Sesssion，所以网站的在线人数是不会变的。



2. 自定义Session扫描器

我们都知道Session是保存在内存中的，如果Session过多，服务器的压力就会非常大。

但是呢，**Session的默认失效时间是30分钟(30分钟没人用才会失效)，这造成Seesion可能会过多（没人用也存在内存中，这不是明显浪费吗？）**

当然啦，我们可以**在web.xml文件中配置Session的生命周期**。但是呢，**这是由服务器来做的，我嫌它的时间不够准确。**（有时候我配置了3分钟，它用4分钟才帮我移除掉Session）

所以，我决定自己**用程序手工移除那些长时间没人用的Session。**

2.1 分析

要想移除长时间没人用的Session，肯定要先拿到全部的Session啦。所以我们**使用一个容器来装载站点所有的Session。。**

只要**Sesssion一创建了，就把Session添加到容器里边**。毫无疑问的，我们需要监听Session了。

接着，我们要做的就是**隔一段时间就去扫描一下全部Session，如果有Session长时间没使用了，我们就把它从内存中移除。隔一段时间去做某事，这肯定是定时器的任务呀。**

定时器应该在服务器一启动的时候，就应该被创建了。因此还需要监听Context

最后，我们还要考虑到并发的问题，如果有人同时访问站点，那么**监听Session创建的方法就会被并发访问了。定时器扫描容器的时候，可能是获取不到所有的Session的。**

这需要我们做同步

于是乎，我们已经有大致的思路了

- **监听Session和Context的创建**
- **使用一个容器来装载Session**

- **定时去扫描Session，如果它长时间没有使用到了，就把该Session从内存中移除。**
- **并发访问的问题**

监听器代码：

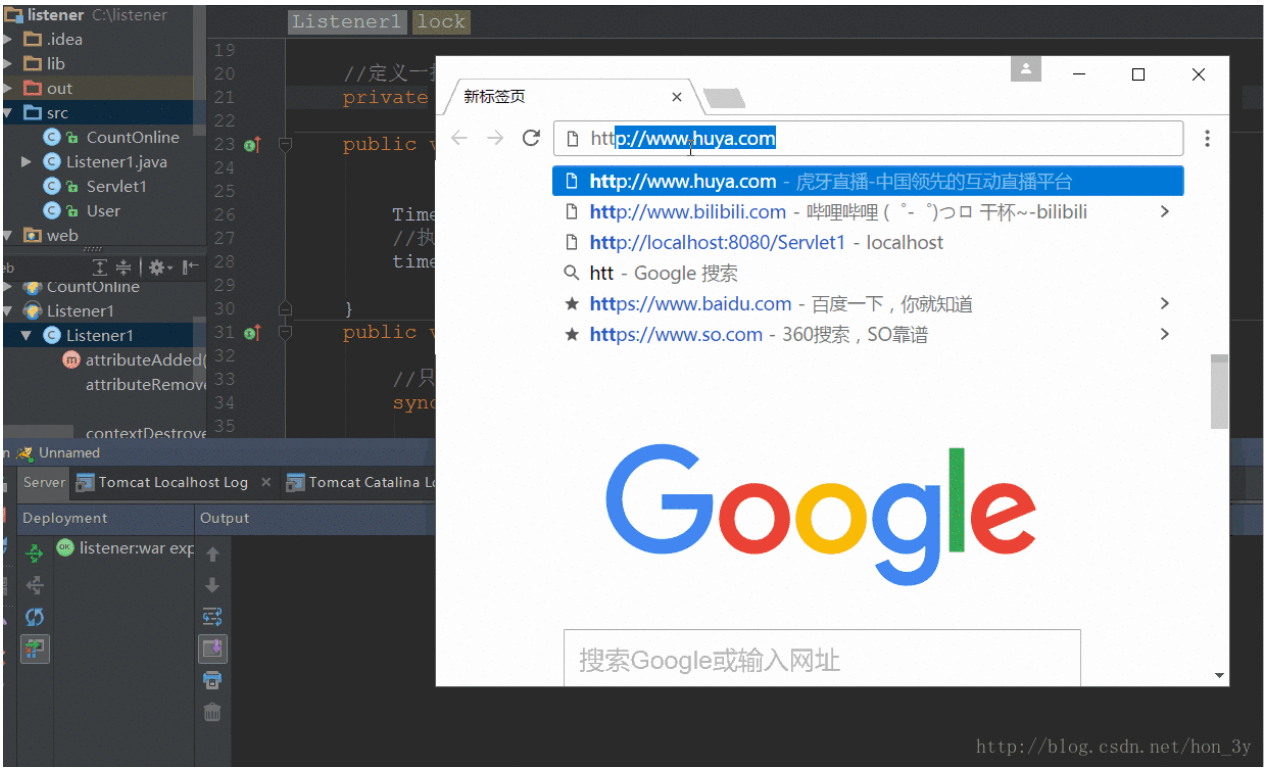
```
  public class Listener1 implements ServletContextListener,
          HttpSessionListener {

      //服务器一启动，就应该创建容器。我们使用的是LinkList(涉及到增删)。容器也应该是线程安全的。
      List<HttpSession> list = Collections.synchronizedList(new LinkedList<HttpSession>());

      //定义一把锁（Session添加到容器和扫描容器这两个操作应该同步起来）
      private Object lock = 1;

      public void contextInitialized(ServletContextEvent sce) {

          Timer timer = new Timer();
          //执行我想要的任务，0秒延时，每10秒执行一次
          timer.schedule(new MyTask(list, lock), 0, 10 * 1000);

      }
      public void sessionCreated(HttpSessionEvent se) {

          //只要Session一创建了，就应该添加到容器中
          synchronized (lock) {
              list.add(se.getSession());
          }
          System.out.println("Session被创建啦");

      }

      public void sessionDestroyed(HttpSessionEvent se) {
          System.out.println("Session被销毁啦。");
      }
      public void contextDestroyed(ServletContextEvent sce) {

      }
  }
```

任务代码：

```
  /*
  * 在任务中应该扫描容器，容器在监听器上，只能传递进来了。
  *
```

```
 * 要想得到在监听器上的锁，也只能是传递进来
 *
 * */
class MyTask extends TimerTask {

    private List<HttpSession> sessions;
    private Object lock;

    public MyTask(List<HttpSession> sessions, Object lock) {
        this.sessions = sessions;
        this.lock = lock;
    }

    @Override
    public void run() {

        synchronized (lock) {
            //遍历容器
            for (HttpSession session : sessions) {

                //只要15秒没人使用，我就移除它啦
                if (System.currentTimeMillis() -
session.getLastAccessedTime() > (1000 * 15)) {
                    session.invalidate();
                    sessions.remove(session);
                }

            }
        }
    }
}
```

15秒如果Session没有活跃，那么就被删除！



- 使用集合来装载我们所有的Session
- 使用定时器来扫描session的声明周期【由于定时器没有session，我们传进去就好了】
- 关于并发访问的问题，我们在扫描和检测session添加的时候，同步起来就好了【当然，定时器的锁也是要外面传递进来的】

---

3. 踢人小案列

列出所有的在线用户，后台管理者拥有踢人的权利，点击踢人的超链接，该用户就被注销了。

3.1 分析

首先，怎么能列出所有的在线用户呢？？一般我们在线用户都是用Session来标记的，**所有的在线用户就应该用一个容器来装载所有的Session。。**

我们监听Session的是否有属性添加(**监听Session的属性有添加、修改、删除三个方法。如果监听到Session添加了，那么这个肯定是个在线用户！**)。

装载Session的容器应该是在Context里边的【属于全站点】，并且容器应该使用Map集合【待会还要通过用户的名字来把用户踢了】

思路：

- 写监听器，监听是否有属性添加在Session里边了。
- 写简单的登陆页面。
- 列出所有的在线用户
- 实现踢人功能(也就是摧毁Session)

监听器

```
public class KickPerson implements HttpSessionAttributeListener {

    // Public constructor is required by servlet spec
    public KickPerson() {
    }

    public void attributeAdded(HttpSessionBindingEvent sbe) {

        //得到context对象，看看context对象是否有容器装载Session
        ServletContext context = sbe.getSession().getServletContext();

        //如果没有，就创建一个呗
        Map map = (Map) context.getAttribute("map");
        if (map == null) {
            map = new HashMap();
            context.setAttribute("map", map);
        }

        //--------------------------------------------------------------------
-------------------

        //得到Session属性的值
        Object o = sbe.getValue();

        //判断属性的内容是否是User对象
        if (o instanceof User) {
            User user = (User) o;
            map.put(user.getUsername(), sbe.getSession());
        }
    }

    public void attributeRemoved(HttpSessionBindingEvent sbe) {
      /* This method is called when an attribute
         is removed from a session.
      */
    }

    public void attributeReplaced(HttpSessionBindingEvent sbe) {
      /* This method is invoked when an attibute
         is replaced in a session.
      */
    }
}
```

登陆页面

```
<form action="${pageContext.request.contextPath }/LoginServlet" method="post">
    用户名: <input type="text" name="username">
    <input type="submit" value="登陆">
</form>
```

处理登陆Servlet

```
        //得到传递过来的数据
        String username = request.getParameter("username");

        User user = new User();
        user.setUsername(username);

        //标记该用户登陆了!
        request.getSession().setAttribute("user", user);

        //提供界面，告诉用户登陆是否成功
        request.setAttribute("message", "恭喜你，登陆成功了！");
        request.getRequestDispatcher("/message.jsp").forward(request,
response);
```

列出在线用户

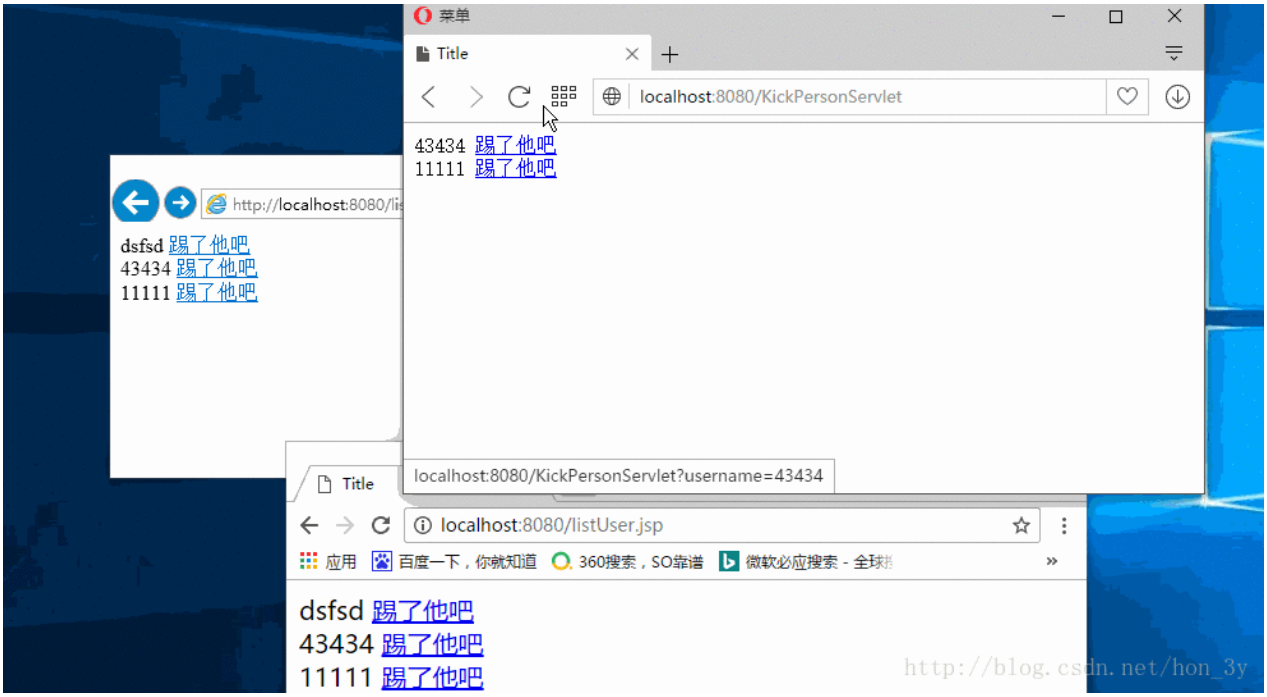
```
<c:forEach items="${map}" var="me">

    ${me.key} <a href="${pageContext.request.contextPath}/KickPersonServlet?
username=${me.key}">踢了他吧</a>

    <br>
</c:forEach>
```

处理踢人的Servlet

```
        String username = request.getParameter("username");

        //得到装载所有的Session的容器
        Map map = (Map) this.getServletContext().getAttribute("map");

        //通过名字得到Session
        HttpSession httpSession = (HttpSession) map.get(username);
```

```
        httpSession.invalidate();
        map.remove(username);

        //摧毁完Session后，返回列出在线用户页面
        request.getRequestDispatcher("/listUser.jsp").forward(request,
response);
```

使用多个浏览器登陆来模拟在线用户（同一个浏览器使用的都是同一个Session）



监听Seesion的创建和监听Session属性的变化有啥区别？？？

- Session的创建只代表着**浏览器给服务器发送了请求。会话建立**
- Session属性的变化就不一样了，**登记的是具体用户是否做了某事(登陆、购买了某商品)**

加油

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

面试题

1. 监听器有哪些作用和用法?

> 监听器有哪些作用和用法?

Java Web开发中的监听器（listener）就是application、session、request三个对象创建、销毁或者往其中添加修改删除属性时自动执行代码的功能组件，如下所示：

- ①ServletContextListener：对Servlet上下文的创建和销毁进行监听。
- ②ServletContextAttributeListener：监听Servlet上下文属性的添加、删除和替换。
- ③HttpSessionListener：对Session的创建和销毁进行监听。
    - 补 充：session的销毁有两种情况：
        - session超时（可以在web.xml中通过`<session- config>/<session-timeout>`标签配置超时时间）；
        - 通过调用session对象的invalidate()方 法使session失效。
- ④HttpSessionAttributeListener：对Session对象中属性的添加、删除和替换进行监听。
- ⑤ServletRequestListener：对请求对象的初始化和销毁进行监听。
- ⑥ServletRequestAttributeListener：对请求对象属性的添加、删除和替换进行监听。

常见的监听器用途主要包括：**网站在线人数技术、监听用户的行为(管理员踢人)。**

2. 过滤器常见面试题

> 过滤器有哪些作用和用法?

Java Web开发中的过滤器（filter）是从Servlet 2.3规范开始增加的功能，并在Servlet 2.4规范中得到增强。对Web应用来说，**过滤器是一个驻留在服务器端的Web组件**，它可以截取客户端和服务器之间的请求与响应信息，并对这些信息进行过 滤。当Web容器接受到一个对资源的请求时，它将判断是否有过滤器与这个资源相关联。如果有，那么容器将把请求交给过滤器进行处理。在过滤器中，**你可以改 变请求的内容，或者重新设置请求的报头信息，然后再将请求发送给目标资源。当目标资源对请求作出响应时候，容器同样会将响应先转发给过滤器，再过滤器中， 你可以对响应的内容进行转换，然后再将响应发送到客户端。**

常见的过滤器用途主要包括：**对用户请求进行统一认证、对用户的访问请求进行记录和审核、对用户发送的数据进行过滤或替换、转换图象格式、对响应内容进行压缩以减少传输量、对请求或响应进行加解密处理、触发资源访问事件、对XML的输出应用XSLT等。**

和过滤器相关的接口主要有：Filter、FilterConfig、FilterChain

3. Java Web常见面试题

3.1web.xml 的作用?

> web.xml 的作用?

答：用于配置Web应用的相关信息，如：**监听器（listener）、过滤器（filter）、 Servlet、相关参数、会话超时时间、安全验证方式、错误页面等。**例如：

①配置Spring上下文加载监听器加载Spring配置文件：

```
<context-param>
   <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>

<listener>
   <listener-class>
     org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

②配置Spring的OpenSessionInView过滤器来解决延迟加载和Hibernate会话关闭的矛盾：

```
<filter>
  <filter-name>openSessionInView</filter-name>
  <filter-class>
     org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>openSessionInView</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

③配置会话超时时间为10分钟：

```
<session-config>
  <session-timeout>10</session-timeout>
</session-config>
```

④配置404和Exception的错误页面：

```
[html] view plaincopy在CODE上查看代码片派生到我的代码片
<error-page>
  <error-code>404</error-code>
  <location>/error.jsp</location>
</error-page>

<error-page>
  <exception-type>java.lang.Exception</exception-type>
  <location>/error.jsp</location>
</error-page>
```

⑤配置安全认证方式：

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>ProtectedArea</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
```

```
</login-config>

<security-role>
  <role-name>admin</role-name>
</security-role>
```

【补 充1】从Servlet 3开始，可以不用在web.xml中部署Servlet（小服务）、Filter（过滤器）、Listener（监听器）等Web组件，**Servlet 3提供了基于注解的部署方式，可以分别使用@WebServlet、@WebFilter、@WebListener三个部署小服务、过滤器、监听器。**

【补充2】如果Web提供了有价值的商业信息或者是敏感数据，那么站点的安全性就是必须考虑的问题。安全认证是实现安全性的重要手段，认证就是要解决“Are you who you say you are?”的问题。认证的方式非常多，简单说来可以分为三类：

A.What you know? --口令

B.What you have? --数字证书（U盾、密保卡）

C.Who you are? -- 指纹识别、虹膜识别

在Tomcat中可以通过建立安全套接字层（Secure Socket Layer, SSL）以及通过基本验证或表单验证来实现对安全性的支持。

4. Servlet 3中的异步处理指的是什么?

> Servlet 3中的异步处理指的是什么?

答： 在Servlet 3中引入了一项新的技术可以让Servlet异步处理请求。有人可能会质疑，既然都有多线程了，还需要异步处理请求吗？答案是肯定的，因为如果一个任务处 理时间相当长，那么Servlet或Filter会一直占用着请求处理线程直到任务结束，随着并发用户的增加，容器将会遭遇线程超出的风险，这这种情况下 **很多的请求将会被堆积起来而后续的请求可能会遭遇拒绝服务，直到有资源可以处理请求为止。异步特性可以帮助应用节省容器中的线程，特别适合执行时间长而且 用户需要得到结果的任务，如果用户不需要得到结果则直接将一个Runnable对象交给Executor（如果不清楚请查看前文关于多线程和线程池的部 分）并立即返回即可。**

开启异步处理代码：

```
@WebServlet(urlPatterns = {"/async"}, asyncSupported = true)
public class AsyncServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // 开启Tomcat异步Servlet支持
        req.setAttribute("org.apache.catalina.ASYNC_SUPPORTED", true);

        final AsyncContext ctx = req.startAsync();  // 启动异步处理的上下文
        // ctx.setTimeout(30000);
```

```
        ctx.start(new Runnable() {

            @Override
            public void run() {
                // 在此处添加异步处理的代码

                ctx.complete();
            }
        });
    }
}
```